

# Application of Genetic Algorithms and Neural Networks to the Solution of Inverse Heat Conduction Problems

## A Tutorial

**Keith A. Woodbury**

*Department of Mechanical Engineering  
The University of Alabama  
Tuscaloosa, Alabama, USA  
woodbury@me.ua.edu*

### ABSTRACT

Genetic Algorithms and Neural Networks are relatively new techniques for optimization and estimation. These techniques can, of course, be applied to the solution of inverse problems. This paper presents a tutorial for application of these techniques to the solution of some simple inverse problems. A description of each of the techniques precedes presentation of the algorithms. MATLAB is used to solve these problems.

### INTRODUCTION

Genetic Algorithms are a class of search methods, which are patterned after evolutionary processes. These algorithms have existed for perhaps 20 years, but were first popularized following the publication of David Goldberg's text on the subject (Goldberg, 1989). These algorithms search a solution space by manipulating populations of candidate solutions. These populations are evaluated to determine the best members of each generation, and each generation reproduces to create the next generation. Notions from evolution are borrowed to manipulate the population after reproduction: randomly triggered crossover and mutation enter in to widen the search region. At the end of a pre-specified number of generations, the results are examined.

A Genetic Algorithm is a type of search procedure. Simply put, it is a localized random search. It requires no evaluation of the derivative of the performance measure, and is therefore highly suited to nonlinear problems.

Neural Networks describe another type of algorithm that is borrowed from nature. Neural Networks are an attempt to model the massively parallel operation of a brain. A collection of simple *neurons* is interconnected with links. Each

link will be assigned a *weight* during the *training* of the network. During operation of the network, the output of each neuron is the result of the weighted sum of all the inputs connected to it passed through an *activation function*. This activation function is some suitable non-linear mathematical function. It is through the sum total action of many connected neurons that the network is able to "learn" its behavior and produce intelligible results.

A Neural Network is an interpolative procedure. Through training, the network "learns" an association between a collection of inputs and their corresponding outputs. In operation, when the network is presented inputs, which were not present in the training set, the network will produce a result, which is consistent with the training data.

This paper has two parts, one devoted to Genetic Algorithms and one dedicated to Neural Networks. In each section a basic description of the method is given, followed by application to a simple optimization problem of parameter identification. Then each method is applied to a classic inverse boundary problem. The MATLAB programming language is used to illustrate the algorithms.

### GENETIC ALGORITHMS

These algorithms mimic the evolutionary processes that have led to development of higher organisms in nature. An initial population of candidates reproduces to create a new generation of the population. In each generation there are random occurrences of mutation in the population. Above all, *survival of the fittest* ensures that the "best" members of the population are retained.

Randomness plays a central role in the genetic algorithm search process. A random number generator will be called thousands of times during the execution of a genetic algorithm.

A “genetic algorithm” is one that possesses the following characteristics:

1. An initial population of a fixed number of candidate solutions is selected.
2. The “fitness” of each of the members of the population is determined using the performance measure for the problem.
3. The members of the current generation of the population reproduce to form the next generation. The reproduction should favor the better members as “parents”. During reproduction, crossover of the genes results in new members not originally in the previous generation but related to them.
4. Random mutation of some of the “children.” These mutations introduce new characteristics not in the previous generation and not directly related to the previous generation but which may result in a “more fit” child.
5. The reproduction continues until a preset number of generations have been created.

At least two types of encoding are possible for the members of the population: binary strings and real number arrays. The latter are more useful for numerical problems, but the former are classic in the genetic algorithms and will be discussed first.

### Binary Encoding

Early applications of genetic algorithms clung to the notion of an organism (member of the population) represented as a *gene* through a binary string of *chromosomes*. The binary string could be interpreted as a color code, and ASCII letter code, an integer code, etc., depending on the problem at hand. But the use of binary encoding facilitates application of the analogy to evolutionary processes.

**Initial Population.** The initial population is typically seeded randomly, but this need not be the case. In the case of binary strings, this could be done bit-by-bit with random selection of a zero or a one for each location.

**Selection of Parents.** Parents are chosen for reproduction based on their fitness. One complicated method for selection of parents is called roulette wheel selection (Davis (1991)), however any method that favors the fitter members of the population may be used.

**Reproduction.** Two Binary strings reproduce through crossover of their chromosomes. After two parents are selected, the child of the two parents is created by splitting the gene (binary string) at one or more points (randomly chosen) and splicing the pieces together.

Figure 1 illustrates reproduction with binary strings. Once the two parents are identified, a crossover point is randomly selected. The Child “AB” results from splicing the first portion of the genetic string “A” onto the second portion of the genetic string “B”. Note that a second child “BA” could easily be produced by splicing the remaining portions of the strings.

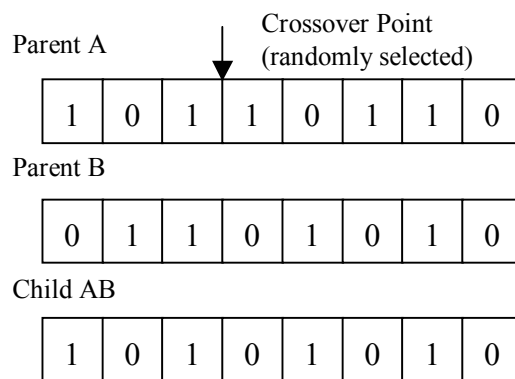


Figure 1. Reproduction through crossover with binary strings.

**Mutations.** After the new generation is created, mutations occur. For each member of the population a random number in the range (0,1) is generated. If the random number is below some prespecified mutation threshold, then the gene is allowed to mutate. Mutation in a binary string is accomplished by selecting one or more (but not all!) of the chromosomes in the string to be altered. Of course “alteration” of a binary digit can only mean changing its sense (flipping it’s bit): if it’s a zero, make it a one, and if it’s a one, make it a zero.

### Real Number Encoding

Binary encoding can literally be applied to any problem at hand. However, if the problem at hand happens to be numerical, and the unknowns of the problem happen to be continuous real numbers, then it is very convenient to represent the members of the population using real number

encoding. A simple example that will be presented below is identification of the slope and intercept of a line based on knowledge of a few of the points on the line. Although these two real numbers (the slope and intercept) most certainly can be represented as a string of binary digits (indeed, in the heart of our computers that is the only form in which they exist!), it is much easier to let the “gene” be an real array of length two. For such a representation we must discover suitable methods for initial population generation, reproduction, and mutation.

**Initial Population.** The initial real number arrays are of course generated randomly. There are at least two possibilities for this and either one could be used, but in any case *the domain of the individual variables* must be known. Remember that the genetic algorithm is only a search mechanism, and the limits of the search region must be known. One possible method of seeding the initial population is to generate a random number for each member of each of the arrays. A second possibility is to generate a random number for each array in the population and assign each array a constant value.

**Selection of Parents.** My technique for selection of parents for reproduction of the next generation is simple. After evaluating the fitness of all the members of the population, sort them in best-to-worst order, then use a specified number of the “best” as parents for the new generation. This assures that the next generation is reproduced using characteristics of the best of the previous generation.

**Reproduction.** The key to reproduction is some sort of crossover mechanism that combines characteristics from each of the parents. In arrays of real numbers both the magnitude of the individual members and the order of the array are important. At least two methods of crossover are possible that address these two characteristics.

To alter the magnitude, averaging of the two parent strings will achieve the desired effect (Davis, 1991). Rather than a simple arithmetic average, I have employed a weighted average with the weight being chosen randomly.

To modify the order of the numbers in each child array, I use a crossover technique identical to that in Fig. 1 for binary strings. Choose a random location in the array, and crossover the sub-arrays from each partner.

**Mutation.** The main purpose of mutation is to introduce new information into the population that can't be obtained directly from the parents.

The method I use was suggested by (Davis, 1991) and is a simple replacement of the array with a randomly generated array within the search space. A mutation threshold must be passed first before the mutation is applied.

**Creep.** This is really a second type of mutation but proves very necessary to refine the search. Creep (Davis, 1991) refers to the drift of the members of a real array around their present value. For each member of the population, a creep threshold is applied, and if the member is eligible to creep, the magnitude of each number in the array is scaled by a random number in the range  $(1 - C, 1 + C)$ , where  $C$  is a fraction between 0 and 1.

**Elitism.** One final mechanism that often is introduced into genetic algorithms is *elitism*. When elitism is employed, the best (or  $N_{elite}$  best) members of each generation are retained in the next generation. This allows the characteristics of the “super-individual” to dominate over several generations.

### A Simple Example

To illustrate the techniques for real number encoding, consider a simple parameter estimation problem. Suppose we have a number  $N_{data}$  of  $(x_i, y_i)$  data pairs and we want to know the equation of a straight line that passes through these points (or close to them):

$$y = b + mx; \quad (1)$$

in other words, the constants  $b$  and  $m$  are to be determined. The classic solution to this problem involves the minimization of the sum of the squared errors between the model-predicted value and the corresponding data value:

$$S = \sum_{i=1}^{N_{data}} (y_i - y)^2 \quad (2)$$

This same methodology will be used to solve this problem using genetic algorithms.

MATLAB was used to code a genetic algorithm to solve this problem, and the main function is shown in Listing 1. Several parameters are passed to the routine: the *xvals* at which the known *ydata* are supplied, the domain of the search (*low*, *high*), which applies to both the slope  $m$  and intercept  $b$ . Other parameters must be specified for the search:  $N_{pop}$  is the number of members of the population;  $N_{best}$  is the number of the best members of the population to used for reproduction at each new generation;  $N_{gen}$  is the number of generations to produces before the program terminates; *mut\_chance* is the

probability threshold for a gene mutation; *creep\_chance* is the threshold for creep of the member and *creep\_amount* is the maximum magnitude of the random creep (the parameter *C* introduced above).

The routine begins with initialization of some arrays and constants. The value *Nelite* is set to one, which means that only the best member of the population is retained from one generation to the next. The *population* is an array of real numbers with *Npop* rows and two columns (one for intercept *b* and one for slope *m*). This array is initialized using uniform random numbers in the range (*low*, *high*) for the intercept and slope.

The main loop of the routine performs the following steps. The model (Eq. (1)) is used to compute the values *ytest* using the function *straight\_line* and all the current members of the population. These *ytest* values are compared to the *ydata* values supplied and a fitness index is computed for each member of the population using Eq. (2). These fitness values are used to sort the population from best to worst, and the *Nbest* members are used for reproduction of the next generation. The reproduction is performed by the routine *reproduce\_by\_weighted\_avg* using the weighted averaging scheme described earlier. Only after new children are produced, the crossover mechanism is applied to all the new members, and the mutation and creep mechanisms are applied randomly based on the thresholds specified.

As a demonstration, the data for a straight line with intercept  $b = 1$  slope  $m = 2$  is used ( $xvals = [1\ 2\ 3\ 4\ 5]$ ; and  $ydata = [3\ 5\ 7\ 9\ 11]$ ). The parameter *mut\_chance* was set to 0.1, meaning that there is a 10% chance that a child will mutate (have its values completely replaced by randomly generated numbers in the domain (*low*, *high*)). The parameters *creep\_chance* and *creep\_amount* were set to 0.90 and 0.25, respectively, meaning that there is a 90% chance that the child will have its value randomly scaled by +/- 25%.

As a first attempt, 50 generations are computed using a population of only 10 members and allowing only the best two for reproduction. At the end of the 50 generations, the best member of the population was  $b = -0.0756$  and  $m = 2.2994$ . The resulting “convergence history”, which is the error of the best member of the population at each generation, is shown in Fig 2. The corresponding estimates for the points on the line are shown in Fig. 3. These results were obtained in 0.22

seconds of CPU time on a 1000 MHz Pentium 4 processor.

The values obtained are not very good. The size of the population is too small to allow the effects of mutation and creep to widen the search. Note that without mutation and creep, the search space will be constrained to that enclosed by the initial randomly generated population as the weighted averaging cannot create a candidate outside that domain.

For the next attempt, the population is increased to 50 and the number of the best to use for reproduction is increased to 10. The number of generations to compute is increased to 100. At the end of the 100 generations, the best member of the population was  $b = 0.9851$  and  $m = 2.0042$ . The convergence history is shown in Fig. 4 and the computed *y* values are compared with the data

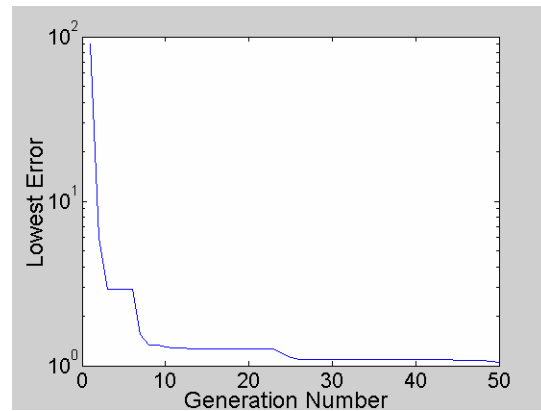


Figure 2. Convergence history for  $Npop = 10$  and  $Nbest = 2$ .

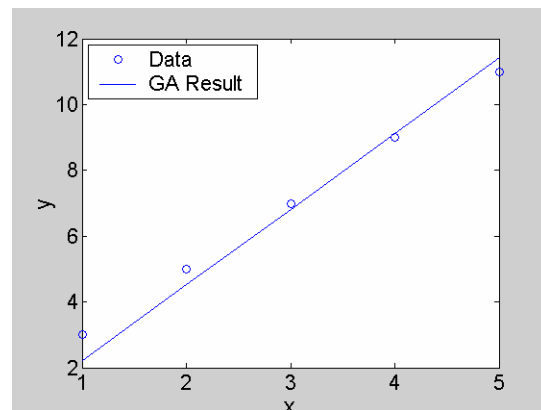


Figure 3. Model estimates (circles) and exact values (line) for  $Npop = 10$  and  $Nbest = 2$ . ( $Ngen = 50$ )

in Fig. 5. These results were obtained in 10.22 CPU seconds on the 1000 MHz Pentium 4 processor.

Note that the results here are much better than those obtained previously, with the minimum sum squared error below  $10^{-3}$ . From the convergence history (Fig. 4) we can see that, after the initial drop, there is little improvement in the lowest error. In fact, a good solution is obtained after 20 generations or so.

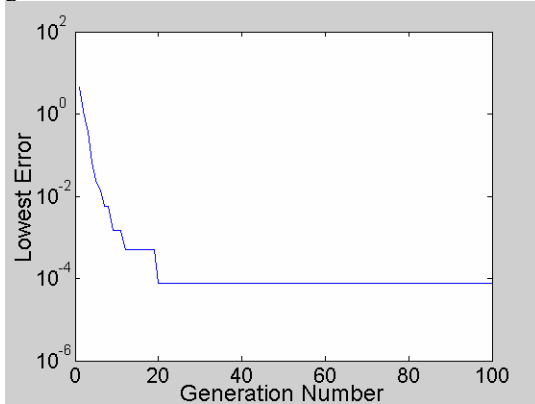


Figure 4. Convergence history for  $N_{pop} = 50$  and  $N_{best} = 10$ .

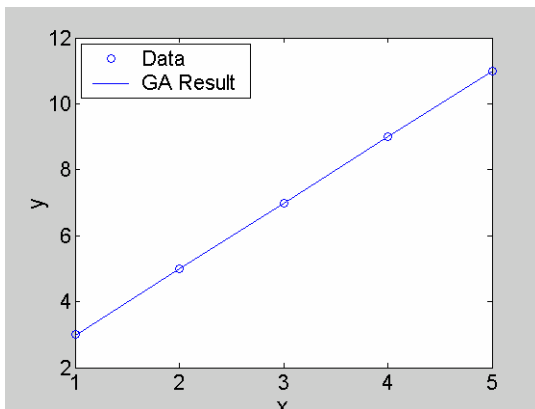


Figure 5. Model estimates (circles) and exact values (line) for  $N_{pop} = 50$  and  $N_{best} = 10$  ( $N_{gen} = 100$ )

### Classic Function Estimation Example

As a much more challenging example, consider the classical problem of estimation of the surface heat flux history for a one dimensional slab insulated at  $x=L$ . This problem was considered previously by Raudensky, et al. (1995), but they worked with an unknown heat transfer coefficient rather than the heat flux. The triangular heat flux history popularized by Beck, et al. (1985) will be used as the “unknown function” to generate data for the estimation

problem. For simplicity, the parameters are taken as  $k = \rho c_p = L = 1$ , which of course is the same as using non-dimensional data. Two cases of data are considered: data with an interval 0.18 s and another with interval 0.06 s. The heat flux history has the following character:

$$q(t) = \begin{cases} 0, & t < 0.24 \\ t - 0.24 & 0.24 \leq t < 0.84 \\ 0.6 - (t - 0.84) & 0.84 \leq t < 1.44 \\ 0 & t \geq 1.44 \end{cases} \quad (3)$$

Data were generated for a sensor located at  $x=L$ , and the data generated for the two data intervals are shown in Table 1 and Table 2.

Table 1. Artificial data for large time interval (0.18 s).

$t, \text{secs}$	$T, C$
0.00	0.00000
0.18	0.00000
0.36	0.00037
0.54	0.01338
0.72	0.05446
0.90	0.12720
1.08	0.21959
1.26	0.29501
1.44	0.34067
1.62	0.35655
1.80	0.35942
1.98	0.35990

**Data Representation.** To estimate the heat flux variation, a suitable parameterization of the heat flux function  $q(t)$  is necessary. We will choose a piecewise constant heat flux, and will estimate one component of heat flux between every temperature data point. So, in the case of the large time interval data of Table 1, there will be 12 unknown heat flux components, and the in case of the small data interval in Table 2, there will be 35 unknown components. (Our algorithm estimates a heat flux component for every data point, even the first one; this assumes a zero temperature initial condition). Again, each member of the population will be a real vector of the appropriate length.

**Function Evaluation.** The objective function for fitness will again be the sum of the squared errors between the model-computed values (objective function for fitness will again

be the sum of the squared errors between the model-computed values ( $y_{test}$ ) and the data ( $y_{data}$ ). The model-computed values will be produced using the Duhamel’s summation as

Table 2. Artificial data for small time interval (0.06 s)

$t, \text{secs}$	$T, C$	$t, \text{secs}$	$T, C$
0.00	0.00000	1.08	0.21959
0.06	0.00000	1.14	0.24768
0.12	0.00000	1.20	0.27293
0.18	0.00000	1.26	0.29501
0.24	0.00000	1.32	0.31371
0.30	0.00001	1.38	0.32895
0.36	0.00037	1.44	0.34067
0.42	0.00217	1.50	0.34882
0.48	0.00632	1.56	0.35376
0.54	0.01338	1.62	0.35655
0.60	0.02366	1.68	0.35809
0.66	0.03732	1.74	0.35894
0.72	0.05446	1.80	0.35942
0.78	0.07515	1.86	0.35968
0.84	0.09939	1.92	0.35982
0.90	0.12720	1.98	0.35990
0.96	0.15788	2.04	0.35995
1.02	0.18929		

described in Chapter 3 of Beck, et al. Note that this computation will be approximate, especially for the larger time intervals.

**Algorithm Description.** The MATLAB main function for this genetic algorithm is shown in Listing 2. Many of the features are the same as in the SimpleGA function, but several enhancements have been made. Specifically, the parameters for the problem ( $N_{gen}$ ,  $mut\_chance$ ,  $creep\_chance$ ,  $creep\_amount$ ) are all vector quantities to facilitate modification of these parameters during the simulation. Also, some regularization has been added via a Tikhonov term, and the coefficients for this are passed through the function call. These enhancements will be described in more detail below.

**Large time interval data.** The case of a larger time step in the data is easier from a classical solution point of view and thus this case will be take first.

As a starting point, use the same parameters that were used at the end of SimpleGA –  $N_{pop} = 50$ ,  $N_{best} = 10$ , and  $N_{gen} = 100$ . After 100 generations, the sum squared error  $S$  is less than  $10^{-3}$ , and the results for the computed temperatures  $y_{test}$  can be seen compared to the data in Fig. 6. The convergence history can be seen in Fig. 7. These results are obtained on a 1.6 GHz Pentium 4 in 5.7 CPU seconds.

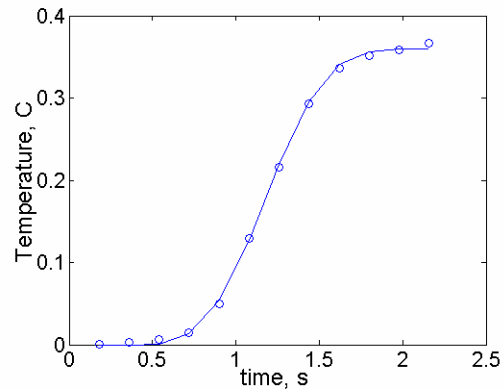


Figure 6. Computed temperatures (circles) compared to data (line) history for first large time interval run.

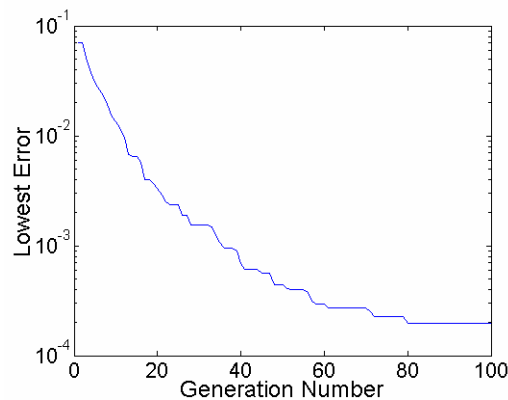


Figure 7, Convergence history for first large time interval run.

The sumsquared error is pretty low, and the computed values of  $T(t)$  are relatively close to the data values (as shown in Fig. 6). But the estimated heat flux components bear little resemblance to the actual input (see Fig. 8). Considering the results (Fig. 8.) and the convergence history, it seems plausible that the problem is not converged well enough. Note that the convergence history decreases past 80 generations, but then levels out. Note also that the history exhibits a mix of large-scale changes (probably caused by mutations) and smaller scale decreases (perhaps brought about by creep). But after 80 generations, the large scale changes (in the domain (-1, 1) and small scale changes (on the order of 25%) are too large to bring any improvement. What is needed is a multiple parameter approach.

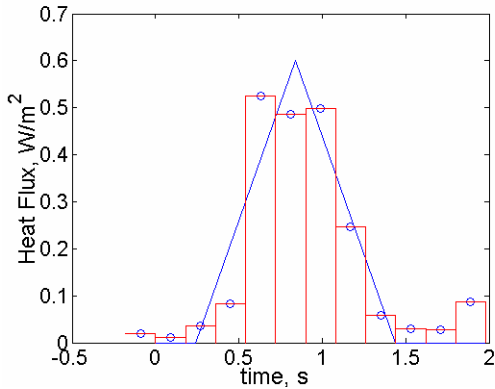


Figure 8. Computed (circles) and actual (line) heat flux input for first large time interval run.

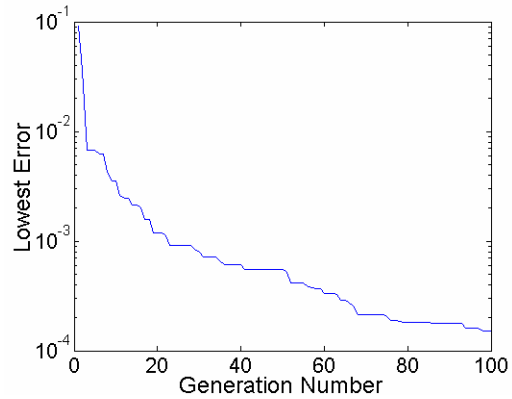


Figure 9. Convergence history for second large time interval run.

A modification to the algorithm allows for this. Next let's try the allowing larger more mutations for the first 50 generations ( $mut\_chance = 0.2$ ), but then decrease the mutation chance to 0.1, and keep the creep chance the same, but decrease the creep amount:  $mut\_chance = 0.1$ ,  $creep\_chance = 0.9$ , and  $creep\_amount = 0.1$ ). The idea is to let the search fine-tune the result after "getting close".

One other modification was made to the program. After each "break" in the generational loop (after the 50 generations, say), the domain for mutations is changed from the initial values to the (minimum, maximum) of the heat flux vector. The idea is to keep the mutation changes within the most reasonable range.

The convergence history, seen in Fig. 9, shows improvement in both the final value and the sustained decrease past 80 generations. The final  $S$  parameter is about  $2 \times 10^{-4}$ , which is pretty good. Considering Fig. 9, more generations may help reduce the error.

The convergence history for several subsequent runs are seen in Fig. 10. The final run corresponds to the lowest line, which achieved an  $S$  parameter of almost  $8 \times 10^{-4}$ . This last run corresponds to a parameter strategy of  $Ngen = [50\ 100\ 200\ 300]$ ,  $mut\_chance = [0.2\ 0.1\ 0.05\ 0.02]$ ,  $creep\_chance = [0.9\ 0.9\ 0.9\ 0.9]$ , and  $creep\_amount = [0.2\ 0.1\ 0.05\ 0.02]$ . The graphical comparison of the computed  $y_{test}$  and the given data  $y_{data}$  is seen in Fig. 11, and it can be seen that the comparison is quite good. The estimates for the heat flux history, seen in Fig. 12, are quite good, and do reproduce the input curve favorably.

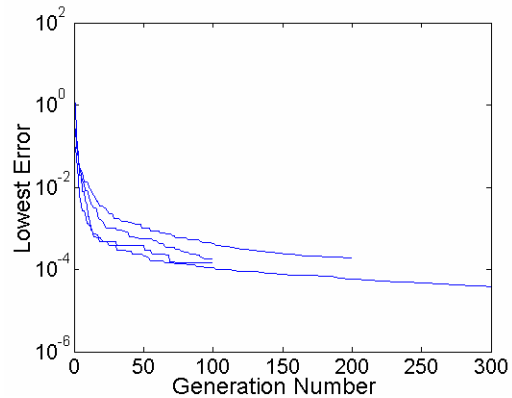


Figure 10. Convergence histories for several subsequent simulations.

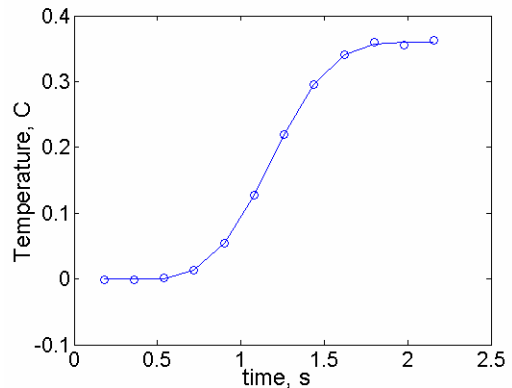


Figure 11. Computed (circles) values of temperature compared with the data (line)

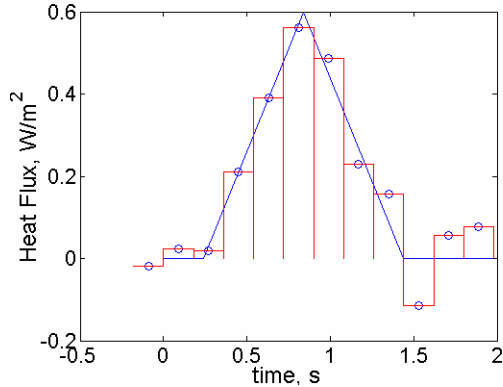


Figure 12. Heat flux estimates for last simulation.

**Small Time Interval Data.** The data from Table 2, which has a (dimensionless) time step of 0.06, is known to cause estimation problems using unregularized methods (such as Stoltz data matching). We next apply the genetic algorithm search to this data.

The same parameters used were the same as in the last run with large time step in the data ( $N_{gen} = [50\ 100\ 200\ 300]$ ,  $mut\_chance = [0.2\ 0.1\ 0.05\ 0.02]$ ,  $creep\_chance = [0.9\ 0.9\ 0.9\ 0.9]$ , and  $creep\_amount = [0.2\ 0.1\ 0.05\ 0.02]$ ). The convergence history (Fig. 13) suggests the results should be good, and the comparison between the computed and measured temperatures confirms this view (Fig. 14). However, the plot of the estimated heat fluxes shows that the underlying ill-posedness of the problem prevents a reasonable result from being obtained (Fig. 15.).

To get a better result in the face of ill-posedness, some regularization must be added to the problem. A familiar Tikhonov regularization term can be added to the objective function (fitness measure) to penalize changes in the first derivative of the heat flux. This is implemented in a discrete form (as described in Chapter 4 of Beck, et al, 1985) as:

$$S = \sum_{i=1}^{N_{data}} (y_i - y)^2 + \sum_{j=1}^{N_{data}-2} \alpha_1 (q_{j+1} - q_j)^2 \quad (3)$$

The  $\alpha_1$  parameter is the regularizing parameter and must be specified.

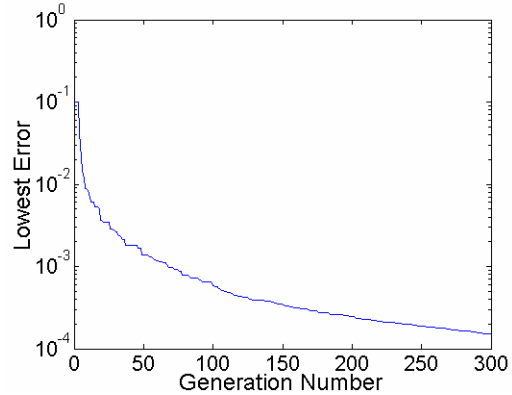


Figure 13. Convergence History for small time step data.

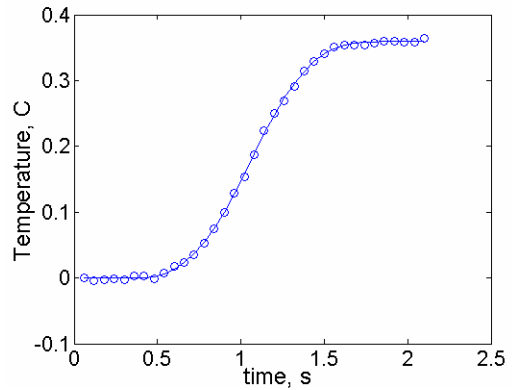


Figure 14. Computed (circles) values of temperature compared with the data (line) for first simulation with small time steps.

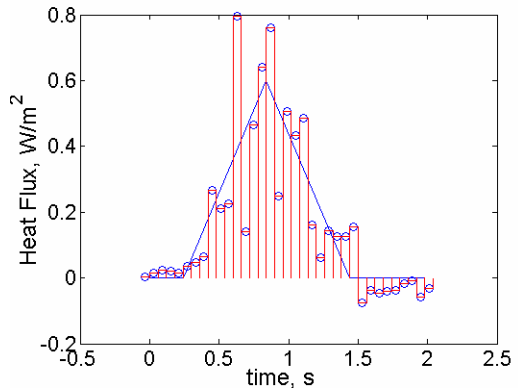


Figure 15. Heat flux estimation for first simulation with small time steps.

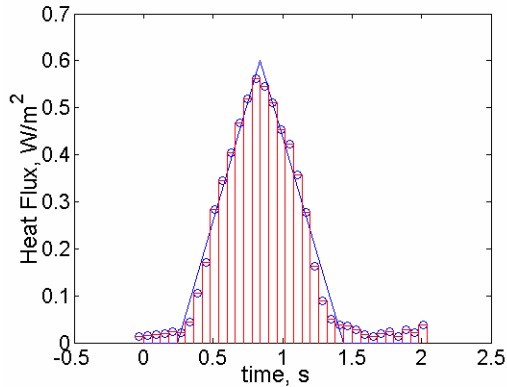


Figure 16. Heat flux estimation with same parameters as Fig. 15, but with Tikhonov regularization ( $\alpha_1 = 1.e-3$ ).

Figure 16 shows the results from a simulation using the same parameters as before, but with a First order Tikhonov regularizing term ( $\alpha_1 = 1.e-3$ ). The converged solution more clearly defines the input heat flux. Note that the algorithm has difficulty where the heat flux is zero (Figs. 8, 12, 15, and 16).

### Genetic Algorithms Conclusions

Genetic algorithms are a random search procedure that search in a fixed domain without using function gradient information. They can be applied to linear or nonlinear problems and are by nature computationally intensive. Real number arrays can be used as “genes” in the population to represent engineering data. Genetic algorithms can be applied to ill-posed problems such as the inverse heat conduction problem, but this solution technique does not evade the inherent ill-posedness of the problem. Some regularization, such as Tikhonov regularization, must be applied in the objective (fitness) function to combat the ill-posedness.

### NEURAL NETWORKS

Neural networks have been used for perhaps 50 years, dating from the early works of Frank Rosenblatt (Rosenblatt, 1961). The main feature of neural networks is in pattern recognition. The network “learns” the relationship between given input and output, and then generalizes this “knowledge”. The result is that when the network is given inputs that are not exactly the same as those from the training data, the output from the network will be something consistent with the

training data. In this way the neural network can be considered an interpolative algorithm.

An early application of Neural Networks was in simple pattern recognition. A classic example is a network designed to “recognize” letters based on a set of optically encoded inputs. A network might be designed and “trained” to identify the letters of the alphabet based on the sense of a  $6 \times 6$  grid of inputs. But if the network was well trained using, say, a Times Roman font, we might expect the network to yield reasonable results if it was shown letters from another font family, such as Ariel.

Application of Neural Networks, then, have two distinct phases: training and simulation. In the training phase, many pairs of inputs and outputs are shown to the network and the weights within the network are adjusted until the network (hopefully) produces the desired output. In the simulation phase, the training algorithm is deactivated, and the network merely computes the output based on the given inputs.

There are many classifications of Neural Networks according to the construction and of the network. Two broad classes are *concurrent* and *recurrent* or *dynamic* networks. Concurrent networks have all their input given at once and the output of the network depends only on these inputs. In contrast, recurrent or dynamic networks receive their inputs sequentially, and the output of some of the neurons in the network are fed back into the input for subsequent computations. In this paper I consider only concurrent Neural Networks.

### Neural Network Topology

A schematic of a Neural Network is shown in Fig. 17. A typical network is composed of one or more *hidden layers* of *neurons* which are interconnected by *weighted* links. The output of each neuron is typically passed through some linear or non-linear *filter* or *activation function*.

A neuron is shown schematically in Fig. 18. The neuron receives inputs from perhaps  $n$  neurons in the previous layer. The output of the summation node is the dot product of the weights  $w_i$  and the value of the inputs:

$$SUM_{out} = \sum_i^n w_i p_i \quad (4)$$

where  $p_i$  is the value of the input ‘ $i$ ’. If an output filter is used on the neuron, then the output of the neuron is the result of the filter on the  $SUM_{out}$ . Several output filter are possible, including

linear, tangent sigmoid, or hyperbolic sigmoid. A sigmoid function has a mathematical character similar to:

$$OUT_j = \frac{1}{1 + \exp(-SUM_{out})} \quad (5)$$

which asymptotically approaches constant values as  $SUM_{out}$  becomes very large or very small.

During the training phase of the network, the network processes given inputs in an attempt to produce given target outputs. The weights of the interconnections are adjusted by an appropriate algorithm to produce the desired outputs. This is

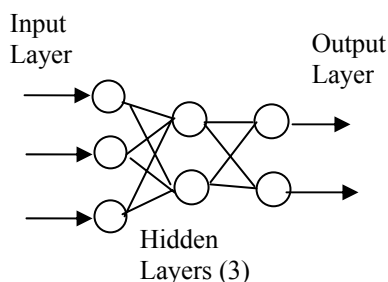


Figure 17. A Schematic of a Neural Network

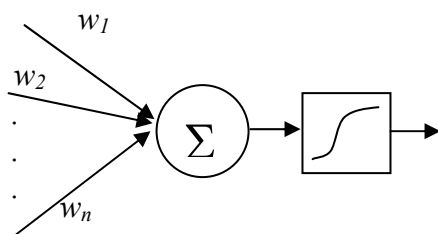


Figure 18. A Schematic illustrating the neuron.

an inherently iterative process, and the number of passes (called *epochs*) through the network during the training may be in the thousands. To train a Neural Network to solve an inverse problem, the mathematical model (forward solution) is used to generate training data.

### MATLAB Toolbox

MATLAB has an excellent toolbox add-in for Neural Network analysis. This collection of programs and interfaces, written by Mark Demuth and Mark Beale, allow easy design and training of a wide range of networks: backpropagation networks, cascade feedforward networks, radial basis function networks, and many recurrent

networks as well. The examples presented here make use of this toolbox add-in.

### A Simple Example

As a simple example, consider the parameter identification problem considered earlier: the estimation of the slope and intercept of a line based on knowledge of several data points. We will design our network to estimate the slope  $m$  and intercept  $b$  of a line over  $0 < x < 1$ , and we restrict the range of  $b$  and  $m$  to the interval  $[0,1]$ . Furthermore, the values of  $y$  at specified  $x$  locations of 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0 will be given to the network to estimate  $b$  and  $m$ .

A backpropagation network with 6 inputs (for the 6 values of  $y$ ) and two outputs (for the two values  $b$  and  $m$ ) is created. One hidden layer with 12 neurons is employed, and a tangent sigmoid is chosen for the activation function on the hidden layer. The input layers have linear activation functions (filters).

A set of training data are generated which covers the solution space. I used the  $b$  and  $m$  pairs in Table 3 to generate  $y$  data on the specified intervals for  $x$ , resulting in 20 training vector pairs: input vectors of length 6 containing the values of  $y$  at the specified locations, and the corresponding output values of  $b$  and  $m$  in vectors of length 2.

Table 3. Intercept and slope used to generate data sets for training.

$b$	$m$	$b$	$m$
0.00	1.00	0.00	0.00
0.25	0.75	0.25	0.25
0.50	0.50	0.50	0.50
0.75	0.25	0.75	0.75
1.00	0.00	1.00	1.00
1.00	0.00	1.00	1.00
0.75	0.25	0.75	0.75
0.50	0.50	0.50	0.50
0.25	0.75	0.25	0.25
0.00	1.00	0.00	0.00

The network was set up and trained in the MATLAB toolbox. A scaled conjugate gradient training method (Demuth and Beale, 2001) was used to adjust the weights in the network. After 3000 epochs in the training, the sum squared error between the network output and the targets was 3.E-7.

After training, the network was tested with the eight vector inputs shown in Table 4. Note that

these vector inputs are not in the training set but do cover the range of inputs used in the training. The actual parameters corresponding to the rows of  $y$  values in Table 4 are shown in Table 5, along with the values computed using the trained Neural Network. As can be seen in Table 5, the values estimated from the Neural Network are reasonably good: the RMS errors are 0.0255 for  $b$  and 0.0069 for  $m$ .

Table 4. Test vectors (in rows) for the Line Identification Neural Network

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$
0.90	1.04	1.18	1.32	1.46	1.60
0.10	0.28	0.46	0.64	0.82	1.00
0.90	0.92	0.94	0.96	0.98	1.00
0.30	0.40	0.50	0.60	0.70	0.80
0.50	0.56	0.62	0.68	0.74	0.80
0.30	0.44	0.58	0.72	0.86	1.00
0.70	0.76	0.82	0.88	0.94	1.00
0.70	0.88	1.06	1.24	1.42	1.60

Table 5. Actual and Computed Line Parameters for the rows in Table 4 using single hidden layer

ACTUAL		NN Output	
$b$	$m$	$b$	$m$
0.9	0.7	0.8533	0.6928
0.1	0.9	0.1001	0.8997
0.9	0.1	0.9003	0.0997
0.3	0.5	0.2766	0.4958
0.5	0.3	0.5140	0.3139
0.3	0.7	0.3001	0.6999
0.7	0.3	0.7000	0.2999
0.7	0.9	0.7476	0.9110

Another Neural Network was constructed and an additional hidden layer with 12 neurons was added. This network was trained using the same data (generated from Table 3) and after 3000 epochs the sum squared error was 2.3E-8. The test data from Table 4 was again shown to the network, and the results in Table 6 were obtained. The RMS errors for this case are 0.0008 for  $b$  and 0.0210 for  $m$ . Note the improvement in the estimates with an additional hidden layer in the network.

Table 6. Actual and Computed Line Parameters for the rows in Table 4 using two hidden layers

ACTUAL		NN Output	
$b$	$m$	$b$	$m$
0.9	0.7	0.9009	0.6601
0.1	0.9	0.0997	0.9003
0.9	0.1	0.9005	0.0997
0.3	0.5	0.3021	0.4871
0.5	0.3	0.4996	0.3170
0.3	0.7	0.3001	0.7000
0.7	0.3	0.6999	0.3001
0.7	0.9	0.6997	0.9384

Another approach to the line parameter identification problem is to try to train the network to learn the relationship between arbitrary groups of  $(x,y)$  data and the parameters  $b$  and  $m$ . We'll try this by adding an extra 6 inputs to the network, corresponding to the  $x$  locations. The network with two hidden layers was trained using the same data as before, but giving the  $x$  values as input also. The network learned this relationship very easily – in 6 epochs the SSE is less than  $10^{-29}$ . Next, the network was tested by generating  $y$  data values for  $x$  values not in the training set: (0.1, 0.3, 0.45, 0.55, 0.7, 0.9). The six  $(x,y)$  data pairs for the eight test lines were input to the trained network, and the results are seen in Table 7. The results are not as good as those obtained previously with fixed  $x$  value inputs.

Table 7. Actual and Computed Line Parameters for the rows in Table 4 using one hidden layer and  $(x,y)$  inputs

ACTUAL		NN Output	
$b$	$m$	$B$	$m$
0.9	0.7	0.92865	0.68718
0.1	0.9	0.14523	0.87785
0.9	0.1	0.93050	0.092988
0.3	0.5	0.33520	0.48794
0.5	0.3	0.53171	0.29446
0.3	0.7	0.33841	0.68561
0.7	0.3	0.73234	0.29215
0.7	0.9	0.73396	0.88615

### Application to Heat Flux Estimation

Time and space constraints do not allow the demonstration of the neural network to the solution of the inverse heat conduction problem.

However, this problem was considered by Krejsa, et. al (1999). In that work they considered two possible approaches: the whole domain estimation problem (as was considered in the genetic algorithm problem earlier, where all heat flux components are estimated simultaneously) and a sequential estimation scheme. Only concurrent neural networks were considered. Their conclusion was that the whole domain method offered the best possibility for solution of the inverse heat conduction problem using concurrent networks. However, I note here that recurrent networks may offer the possibility of sequential estimation.

The approach to solving the whole domain estimation of heat fluxes is similar to that taken in the line parameter estimation problem. Training data must be generated over the whole solution space of  $(t, q)$ . This might be done by considering a range of different types of inputs: linear ramps of different slope, steps, parabolas, etc. The key is that the generated training data must cover the whole space of possible inputs for the neural network.

### **Neural Network Conclusions**

Neural networks offer the possibility of solution of parameter estimation problems and also boundary inverse problems. Proper design of the network itself *and* the training data set is essential for successful application of this approach.

### **REFERENCES**

1. D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, Mass.: Addison Wesley (1989)
2. L. Davis (ed.), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold (1991)
3. M. Raudensky, K. A. Woodbury, J. Kral, and T. Brezina,, "Genetic Algorithm in Solution of Inverse Heat Conduction Problems," Numerical Heat Transfer, Part B: Fundamentals, Vol 28, no 3, Oct.-Nov. 1995, pp. 293-306.
4. J. V. Beck, B. Blackwell, and C. St. Clair, *Inverse Heat Conduction: Ill-posed problems*, New York: John Wiley (1985).
5. F. Rosenblatt, *Principles of Neurodynamics*, Washington, DC: Spartan Books (1961).
6. H. Demuth and M Beale, *Neural Network Toolbox: For Use with MATLAB (User's Guide)*, Natick, MA: The Mathworks (2001)

7. J. Krejsa, K. A. Woodbury, J. D. Ratliff., and M. Raudensky, "Assessment of Strategies and Potential for Neural Networks in the IHCP," Inverse Problems in Engineering, Vol 7, n 3, pp. 197-213. (1999)

Listing 1. Main function for the estimation of parameters of a straight line

```

% function simpleGA( xvals, ydata, Npop, low, high, Nbest, Ngen, mut_chance, creep_chance,
    creep_amount )
% function to perform simple GA search
% find the slope and intercept of a line described the
% data in 'xvals' and 'ydata'
% 1) randomly initialize 'Npop' candidate vectors in range
%   'low' to 'high'
% 2) evaluate the fitness using least squares criteria
% 3) sort to find best 'Nbest' candidates to use for reproducing
% 4) repopulate using random selection from the Nbest and random recombination
%   (use elitism - keep the top performer)
% 5) allow mutation of new generation at 'mut_chance' rate of magnitude
%   mut_amount
% 6) repeat for 'Ngen' new generations
function simpleGA( xvals, ydata, Npop, low, high, Nbest, Ngen, mut_chance, creep_chance,
    creep_amount )
Nunknown = 2; % slope and intercept are the two unknowns
Nelite = 1; % clone the super-individual
Ndata = length( ydata );

% generate the initial population
population = gen_rand_real( Npop, Nunknown, low, high );
% create matrix to keep best values and fitness for each generation
best = zeros( Ngen, Nunknown + 1 );
index = zeros( Ngen, 1 );

% loop for the generations
for gen = 1 : Ngen
    % use model to compute values from population
    ytest = straight_line( population, xvals );
    % compute the fitness using least squares
    fitness = sum_square_fitness( ytest, ydata );
    % sort them
    sorted = sort_by_fitness( fitness, population );
    % copy generation champion into storage array
    best( gen, : ) = sorted( 1, : );
    index(gen)=gen;
    % copy best candidates into top of population
    population( 1:Nbest, : ) = sorted( 1:Nbest, 2:Nunknown+1 );
    % reproduce to fill the bottom of the population
    population = reproduce_by_weighted_avg( Nbest, population, Nelite );
    % crossover the children
    population = crossover( Nelite, population );
    % creep the children
    population = creep( Nelite, population, creep_chance, creep_amount );
    % mutate the children
    population = mutate( Nelite, population, mut_chance, low, high );
end
figure(1);
semilogy( index, best(:,1) );
b_best = best( Ngen, 2 );

```

```

m_best = best( Ngen, 3);
ybest = b_best * ones( 1, Ndata ) + m_best * xvalls;
figure(2);
plot( xvalls, ydata, '-' );
hold on;
plot( xvalls, ybest, 'o' );
hold off;
best
sorted;
population;

```

Listing 2. Main Function for Estimating Heat Flux History

```

% function estimateQ_GA( xvalls, ydata, dt, Nunknown, Npop, low, high, Nbest, Ngen, mut_chance,
    creep_chance, creep_amount, alpha )
% function to estimate a heat flux function of time using simple GA search
% the location(s) of the sensors are contained in 'xvalls', and the corresponding
% measurements vectors are in 'ydata' (for more than one location xvalls(1),
% xvalls(2), etc., the data are ydata = [ ydata(1), ydata(2), etc. ].
% The time step in the data is 'dt'
% The vector 'alpha' is length 2 and contains the tikhonov regularization
% scalar weights.
%
% 1) randomly initialize 'Npop' candidate vectors in range
% 'low' to 'high'
% 2) evaluate the fitness using least squares criteria
% 3) sort to find best 'Nbest' candidates to use for reproducing
% 4) repopulate using random selection from the Nbest and random recombination
% (use elitism - keep the top performer)
% 5) allow mutation of new generation at 'mut_chance' rate of magnitude
% mut_amount
% 6) repeat for 'Ngen' new generations
function estimateQ_GA( xvalls, ydata, dt, Nunknown, Npop, low, high, Nbest, Ngen, mut_chance,
    creep_chance, creep_amount, alpha )
nx = length(xvalls);
Nloop = length( Ngen );
Ndata = round( length(ydata) / nx ); % number of values in the unknown vector
Nelite = 5; % clone the super-individuals

% generate the initial population
% for random distribution
% population = gen_rand_real( Npop, Nunknown, low, high );
% for uniform distribution
values = gen_rand_real( Npop, 1, low, high );
population = ones( Npop, Nunknown);
for i = 1:Npop
    population( i, : ) = population( i, : ) * values(i);
end

% create matrix to keep best values and fitness for each generation
best = zeros( Ngen(Nloop), Nunknown + 1 );
index = zeros( Ngen(Nloop), 1);

t_x = [ dt xvalls ]; % special data vector for the evaluation function

```

```

% loop for the generations
gen = 1;
for loop = 1 : Nloop
    for gen = gen : Ngen(loop)
        % use model to compute values from population
        ytest = eval_qvec( population, t_x, Ndata );
        % compute the fitness using least squares
        fitness = sum_square_fitness( ytest, ydata );
        fitness = fitness + tikhonov_term( population , alpha );
        % sort them
        sorted = sort_by_fitness( fitness, population );
        % copy generation champion into storage array
        best( gen, : ) = sorted( 1, : );
        index(gen)=gen;
        % copy best candidates into top of population
        population( 1:Nbest, : ) = sorted( 1:Nbest, 2:Nunknown+1 );
        % reproduce to fill the bottom of the population
        population = reproduce_by_weighted_avg( Nbest, population, Nelite );
        % crossover the children
        population = crossover( Nelite, population );
        % creep the children - modify slightly each value (by chance)
        population = creep( Nelite, population, creep_chance(loop), creep_amount(loop) );
        % mutate the children - random replace of chromosome
        population = mutate( Nelite, population, mut_chance(loop), low, high );
    end
    range = minmax( best( gen, 2:Nunknown+1 ) )
    low = range(1);
    high = range(2);
end
figure(1);
semilogy( index, best(:,1) );
time_max = Ndata * dt;
dt_unk = time_max / Nunknown;
time = zeros( Ndata, 1);
time_half = zeros( Nunknown, 1);
time(1) = dt;
time_half(1) = dt_unk/2;
for i = 2:Nunknown
    time_half(i) = time_half(i-1) + dt_unk;
end
for i = 2:Ndata
    time(i) = time(i-1) + dt;
end
last = best( Ngen(Nloop), 2:Nunknown+1);
figure(2);
plot( time_half, last, 'o' );
t_exact = [ 0 .24 .84 1.44 1.8 ];
q_exact = [ 0 0 0.6 0 0 ];
hold on;
plot( t_exact, q_exact, '-' );
hold off;
qbest = best(Ngen(Nloop),2:Nunknown+1)
% use model to compute values from population

```

```
ybest = eval_qvec( qbest, t_x, Ndata );  
figure(3);  
plot( time, ybest, 'o' );  
hold on;  
plot( time, ydata, '-' );  
hold off;  
sum_sq_err = sum_square_fitness( ybest, ydata );  
rms_error = sqrt(sum_sq_err/Nunknown)  
sorted;  
population;
```